Basic Programming Concepts

- Variables and Expressions
- Conditional Statements
- Loops
- Collections (Arrays and ArrayLists)

Recall From Last Time...

- A class contains two types of elements:
  - Instance data — variables that represent the information held by that class
  - Instance methods — methods (code) that define what a class can do
- A given class does not have to contain both elements
  - A class may contain only data, or only methods

Variables

- Variables are the “nouns” of a program
- They represent pieces of information
- Variables come in different types:
  - int and long store integers
  - char stores single characters (‘a’)
  - float and double store decimal values
Declarations

- Java requires you to declare a variable before you use it
- This tells Java what behavior to expect from that variable
- Declare a variable (tell the compiler that you’re going to use it) by writing the type, followed by the name:
  
  int x; // Declares an integer variable x
doouble pi; // pi holds a fractional number

Storing Values in Variables

- Use a single ‘=’ to store a value in a variable
- “=” translates to “is assigned the value”
- The left gets the value on the right side:
  
  x = 5; // Stores the integer value 5 in x
- We can combine assignments with declarations:
  
  char foo = ‘g’;
- 5 and ‘g’ are literals (actual values)

Constants

- A constant is a value that cannot change
- To declare a constant, use the “final” keyword:
  
  final double PI = 3.1415926;
- By convention, constant names are capitalized
- Strings (sequences of characters enclosed in double quotes) are automatically constants.

Arithmetic

- Basic arithmetic (addition, subtraction, multiplication, division) uses the normal order of operations
- Dividing two integers only gives you the quotient
- Use modulus (%) to get the remainder
  
  e.g., 12 / 7 is 1, and 12 % 7 is 5
- The modulus operation works like “clock arithmetic”
The Rules of Arithmetic

- In Java, arithmetic follows two basic rules:
  1. Both operands must be the same type
  2. The answer has the same type as the operands
- If the operands are of different types (e.g., int and float), then they must be converted to the same type first
  - Java automatically expands smaller types to larger types

boolean << char << int << long << float << double

Type Conversion

- A value can always be stored in a variable of a larger (further right) type
  - Ex. an int can be stored in a float
- If you want to go the other way (to store a value in a smaller type), you must explicitly cast (convert) the value to the desired type:
  int x = (int)3.14159;
- To cast, precede the value with the new type in parentheses
- Casting may (and often does) lose data; the original value will be truncated (for example, a decimal will lose the part after the point).
- Note: Casting a value to an integer does not round the resulting value

Conversion Examples

double average = 100.0/8.0
average = 100.0/8
average = 100/8

100.0 and 8.0 are both doubles; the result (12.5) is the same type as average.
100.0 and 8 are different types; 8 is upcast to 8.0, and we proceed as before.
100 and 8 are both integers, so we perform integer division to get 12. This is upcast to 12.0, because average is of type double.

Conversion Examples

int sumGrades = 100.0/8.0
sumGrades = (int)100.0/8.0
sumGrades = (int)(100.0/8.0)

100.0 and 8.0 are both doubles; their quotient (12.5) is also a double. The double-to-int cast must be explicit, so the compiler will report an error.
100.0 (a double) is cast to an int, but 8.0 is still a double. 100 is upcast to 100.0, and we have the same problem as before.
In this case, 100.0 and 8.0 are divided first. The result (12.5) is cast from a double to an int, losing the .5, and is stored in subGrades as 12 (no rounding!).
Conversion Examples

double fiftyPercent = 50/100;
fiftyPercent = 50.0/100.0;

50 and 100 are both integers; dividing them produces 0. 0 is converted to a double to match fiftyPercent's type, and the result is 0.0.

50.0 and 100.0 are both doubles; dividing them gives 0.5 (a double). This is stored unchanged in fiftyPercent.

Conditionals and Decision Statements

ISE 208: Intermediate Programming
SUNY at Stony Brook

Making Decisions

- Many programs must behave differently when presented with different situations
  - e.g., choosing an action from a menu, determining what sales tax rate to charge, etc.
  - Java offers two similar ways to handle these situations:
    - branching (if-else) statements
    - selection (switch) statements

True and False Values

- A boolean expression is one whose value is either true or false (but not both!)
  - Ex. a > b or var1 == var2
  - Boolean (logical) equality operator: ==
  - Common programming error: using ‘=' (assignment) instead of “==” (equality)
  - Ex. if (x = 5)
Boolean Operations

- Decisions are made in Java programs using boolean logic (i.e., values of true and false) and operations.
- Relational operators are used to compare two values.
- Logical operators combine smaller boolean expressions into a single, larger expression.

Relational Operators

<table>
<thead>
<tr>
<th>Operator</th>
<th>Meaning</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;</td>
<td>Less than</td>
<td>age &lt; 30</td>
</tr>
<tr>
<td>&gt;</td>
<td>Greater than</td>
<td>height &gt; 6.2</td>
</tr>
<tr>
<td>&lt;=</td>
<td>Less than or equal</td>
<td>taxable &lt;= 20000</td>
</tr>
<tr>
<td>&gt;=</td>
<td>Greater than/equal</td>
<td>temp &gt;= 98.6</td>
</tr>
<tr>
<td>==</td>
<td>Equal to</td>
<td>grade == 100</td>
</tr>
<tr>
<td>!=</td>
<td>Not equal to</td>
<td>number != 250</td>
</tr>
</tbody>
</table>

Logical Operators

- && (logical AND)
  - true only if BOTH operands are true
- || (logical OR)
  - true if AT LEAST ONE operand is true (inclusive OR)
- ! (logical NOT)
  - true if operand is false and vice versa

The if Statement

- General form:
  ```java
  if ( condition )
      statement (or block of statements) to be executed
  if condition is true
  ```

- Ex.
  ```java
  if (length < 2)
      System.out.print("Too short!\n");
  ```
Examples

```java
if (age < 16)
    System.out.print("Too young to drive!\n");

if (password.equals("foo"))
    accessGranted = true;
```

The **if-else** Statement

- Select one of two possible execution paths, based on the result of a comparison
- General format:
  ```java
  if (expression)
      statement block 1
  else
      statement block 2
  ```

More on if Statements

- An if statement executes its body when (and only when) the condition is true
- If the condition is false, the body is skipped, and execution picks up at the first statement after the if
- By default, the body of an if statement is restricted to the first statement that follows the “if (condition)” line
- Indentation doesn’t matter
- This can lead to trouble...

Empty Statements

- A semicolon by itself is a valid (but non-functional) statement
- Common mistake: putting a semicolon immediately after an if statement:
  ```java
  if (x > 5);
      System.out.print("x greater than 5!");
  ```
- With the semicolon, the print statement will execute regardless of the value of x
Compound Statements

- if and else only execute a single following statement
- We can get around this by enclosing multiple statements in braces
- The resulting block is called a compound statement
- Style suggestion: always use braces around the body of an if or else clause

```java
if (hours > 40)
{
    hours = hours - 40;
    overtimePay = hours * 12.0;
    totalPay = (40 * 8.0) + overtimePay;
}
```

Nested if Statements

- A statement block may contain another if statement
- Ex.
  ```java
  if (income > 25000)
      if (deductions < 3500)
          tax_rate = 1.035;
  ```

if-else Chains

```java
if (expression_1)
    statement_1
else if (expression_2)
    statement_2
else
    statement_3
```
The **switch** Statement

```java
switch (variable) {
    case value : statement(s)
    case value : statement(s)
    ...
}
```

**switch** Example

```
switch (labSection) {
    case 1: System.out.print("Section 01\n");
            break;
    case 2: System.out.print("Section 02\n");
            break;
}
```

**switch** Execution

1. The integer expression is evaluated
2. Execution jumps to the **case** statement whose label matches the expression value
3. Execution continues until a **break** statement is encountered (or the **switch** ends)
   - Common error: failure to include **break**

The **default** Case

- The **default** label denotes statements to be executed if no case matches the expression value
- **default** is usually the last label in a **switch** statement
- **Ex.** default: statement(s)
Iteration Statements (Loops)

ISE 208: Intermediate Programming
SUNY at Stony Brook

Iterative Programming

- Many programs perform the same task many times
- Operations are repeated on different data
- Ex. Adding a list of numbers
- Ex. Displaying frames of a movie file
- Repetitive tasks are specified using loops

Loop Elements

All loop constructs share four basic elements:
1. Initialization
2. Testing the loop condition
3. Loop body (the task to be repeated)
4. Loop update

The relative order of these elements may vary

Initialization

- This section of code is used to set starting values
- For example, setting a total to 0 initially
- This can be done as part of the loop, or separately before the loop code begins
Loop Tests

- Test expressions are used to determine whether the loop should execute (again)
- Tests compare one value/variable with another
- If the test evaluates to TRUE, then the loop will execute another time

Loop Update

- This step changes the value(s) of the loop variable(s) before the loop repeats
- Ex. moving to the next item to process
- This can be done explicitly as part of the loop, or it can be done inside the loop body

while Loops

- while loops execute as long as the test condition is true
- Order of execution:
  1. Initialization
  2. Loop condition test
  3. Loop body
  4. Loop update

General Form

\[
\text{initialization}
\]

\[
\text{while ( loop condition test )}
\]

\[
\{ \\
\quad \text{loop body} \\
\quad \text{loop update} \\
\}
\]
while Loop Example

```java
int countDown = 5;
while (countDown >= 0)
{
    System.out.print(countDown);
    System.out.println("...");
    countDown--;
}
```

Loop Breakdown

```
int countDown = 5;               // Loop initialization
while (countDown >= 0)          // Loop test
{
    System.out.print(countDown);  // Loop body
    System.out.println("...");
    countDown--;                  // Loop update
}
```

Loop Output

```
5...
4...
3...
2...
1...
0...
```

Another Example

```java
int root = 0;
while (root < 10)
{
    root += 1;
    System.out.print(root + " * " + root);
    System.out.print(" = ");
    System.out.println(root * root);
}
```
Loop Output

1 * 1 = 1
2 * 2 = 4
3 * 3 = 9
4 * 4 = 16
...
8 * 8 = 64
9 * 9 = 81
10 * 10 = 100

do...while Loops

- Like while loops, but always execute at least once
- Order of execution:
  1. Initialization
  2. Loop body
  3. Loop update
  4. Loop condition test

General Form

```java
initialization
do
{
    loop body
    loop update
} while ( loop condition test ) ;
```

do...while Example

```java
// print numbers from 1-15
int counter = 1;
do
{
    System.out.println(counter + "");
    counter++;
} while (counter < 16);
```
Loop Breakdown

// print numbers from 1-15

int counter = 1;  // Loop initialization

do
{
    System.out.println(counter);  // Loop body
    counter++;  // Loop update
} while (counter < 16);  // Loop test

---

Loop Output

1  2  3  4  5  ...
13 14 15

---

do...while Example

int sum = 0, value = 0;
Scanner in = new Scanner(System.in);
do
{
    sum += value;
    System.out.print("Enter a # (-1 to quit): ");
    value = in.nextInt();
} while (value != -1);

---

for Loops

- for loops execute a fixed number of times
- Order of execution:
  1. Initialization
  2. Loop condition test
  3. Loop body
  4. Loop update
General Form

for ( initialization ;
    loop condition test ;
    loop update )
{
    loop body
}

for Loop Example

for (int i = 0; i < 10; i++)
{
    System.out.println(i);
}

Loop Breakdown

for (int i = 0; i < 10; i++)
{
    System.out.println(i);
}

Loop Output

0
1
2
3
4
5
6
7
8
9
Another Example

```java
int nextNumber, sum = 0;
Scanner in = new Scanner(System.in);
for (int i = 0; i < 5; i++)
{
    System.out.print(“Enter a number: ”);
    nextNumber = in.nextInt();
in.nextLine(); // consume extra newline
    sum += nextNumber;
}
```

<table>
<thead>
<tr>
<th>i</th>
<th>nextNumber</th>
<th>sum</th>
</tr>
</thead>
<tbody>
<tr>
<td>-</td>
<td>-</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>1</td>
<td>15</td>
<td>17</td>
</tr>
<tr>
<td>2</td>
<td>5</td>
<td>22</td>
</tr>
<tr>
<td>3</td>
<td>7</td>
<td>29</td>
</tr>
<tr>
<td>4</td>
<td>3</td>
<td>32</td>
</tr>
<tr>
<td>5</td>
<td>-</td>
<td>32</td>
</tr>
</tbody>
</table>

Fancier `for` Loop Headers

- We can include multiple initialization or update statements in a loop header
- Separate each statement with a comma
- Ex. for (i = 0, j = 1; i < 5; i++, j--) {}
- Loop update statements don’t have to increase by 1
- Ex. for (i = 1; i < 101; i = i + 5) {}

Advanced `for` Loop Headers

- Loop headers can also include calls to methods
  - Ex. for (int i = 0; i < numPlayers(); i++)
- A for loop can also omit part of the header
  - Ex. for (; i < 5; i++) {}
- Pathological example: for (; ; ) {}
Choosing a Loop Type

- For a fixed number of iterations:
  - for loops are generally considered the way to go
- For a variable number of iterations:
  - while loops can execute 0 or more times
  - do...while loops execute at least once
- However, any loop can be rewritten as any other type

Nested Loops

- The body of a loop can contain any other type of statement(s)
  - This includes other loops
- If the outer loop executes \( n \) times, and the inner loop executes \( m \) times, the body of the inner loop will execute \((n \times m)\) times

Nested Loop Example

```java
for (int i = 0; i < 4; i++)
{
    for (int j = 0; j < 4; j++)
    {
        System.out.print("*");
    }
    System.out.print("\n");
}
```

Example Output

```
*****
*****
*****
*****
```
Another Example

```java
int numStars = 1;
do {
    for (int i = 0; i < numStars; i++)
        System.out.print("*");
    System.out.print("\n");
    numStars++;
} while (numStars < 11);
```

Loop Output

```
*
**
***
****
*****
******
*******
********
*********
**********
```

Arrays and ArrayLists

ISE 208: Intermediate Programming
SUNY at Stony Brook

Arrays

- Programs often use large quantities of similar data
- Assigning a unique variable (and name) to each piece of data is tedious
  - Ex. var1, var2, var3, ...
- An array is a collection of many variables of the same type, all under one name
  - Arrays can be of any type
Array Storage

int[] a = new int[6];

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Declaring An Array

- To declare an array, follow the type with square brackets:
  
double[] foo;

- To create an array, use `new` and a size:
  
  foo = new double[5];

  - Array sizes must be non-negative integer values!

Array Size

- The size of an array is fixed upon creation
- The `length` field returns an array's size

```java
int[] a = new int[5];
int size = a.length; // size is 5
```

- Note that `length` has no parentheses!
- It's a variable, not a method

Examples

- Definitions:
  
  ```java
  char[] c;
  int[] value = new int[10];
  ```

- End Result:
  
  - Array object variable `c` is un-initialized
  - Array object variable `v` references a new ten-element list of integers
  - Each of the integers is initialized to 0 by default
Array Examples

```java
char[] alphabet = new char[26];
int numPlayers = 5;
int[] scores = new int[numPlayers];
String[] phrases = new String[15];
```

Creating Arrays

- Arrays can be initialized at declaration:
  ```java
  int[] bar = {5, 4, 3, 2, 1};
  ```
- The size of the array is determined by the size of the initialization list
- In this example, `bar` has five elements

Array Elements

- Individual elements of an array are accessed by using the array name, followed by an (integer) index value, enclosed in brackets
  ```java
  Ex. myArray[1]
  ```
- Indices are numbered starting with 0
  ```java
  Thus, myArray[1] refers to the second element in myArray
  ```
Consider

```java
int[] v = new int[10];
int i = 7;
int j = 2;
int k = 4;
v[0] = 1;
v[i] = 5;
v[j] = v[i] + 3;
v[j+1] = v[i] + v[0];
v[v[j]] = 12;
System.out.println(v[2]);
v[k] = stdin.nextInt();
```

Consider

```java
int[] v = new int[10];
int i = 7;
int j = 2;
int k = 4;
v[0] = 1;
v[i] = 5;
v[j] = v[i] + 3;
v[j+1] = v[i] + v[0];
v[v[j]] = 12;
System.out.println(v[2]);
v[k] = stdin.nextInt();
```
Consider

```java
int[] v = new int[10];
int i = 7;
int j = 2;
int k = 4;
v[0] = 1;
v[i] = 5;
v[j] = v[i] + 3;
v[j+1] = v[i] + v[0];
v[v[j]] = 12;
System.out.println(v[2]);
v[k] = stdin.nextInt();
```

```
1 0 8 6 0 0 0 5 0 0
```

8 is displayed

Suppose 3 is read in
Consider

• Segment
  ```java
  int[] b = new int[100];
  b[-1] = 0;
  b[100] = 0;
  ```

• Causes
  – Array variable to reference a new list of 100 integers
    • Each element is initialized to 0
  – Two exceptions to be thrown
    • -1 is not a valid index – too small
    • 100 is not a valid index – too large
      – IndexOutOfBoundsException

Consider

```java
Point[] p = new Point[3];
p[0] = new Point(0, 0);
p[1] = new Point(1, 1);
p[2] = new Point(2, 2);
p[0].setX(1);
p[1].setY(p[2].getY());
Point vertex = new Point(4,4);
p[1] = p[0];
p[2] = vertex;
```
Consider

```java
Point[] p = new Point[3];
p[0] = new Point(0, 0);
p[1] = new Point(1, 1);
p[2] = new Point(2, 2);
p[0].setX(1);
p[1].setY(p[2].getY());
Point vertex = new Point(4, 4);
p[1] = p[0];
p[2] = vertex;
```
Arrays and Loops

Loops (especially for loops) are the perfect way to manipulate arrays:

```java
int [] a = new int[5];
for (int i = 0; i < a.length; i++)
{
    a[i] = i * 2;
}
```

Fibonacci Numbers

- Special sequence of integers
- First and second numbers are 0 and 1, respectively
- Each subsequent number is the sum of the two previous Fibonacci numbers
  - Ex. Fib(2) = Fib(1) + Fib(0)

```java
int range = sc.nextInt();
int [] fib = new int[range];
fib[0] = 0;
fib[1] = 1;
for (int i = 2; i < range; i++)
{
    fib[i] = fib[i-1] + fib[i-2];
}
// fib[n] contains the nth Fibonacci number
```

Multidimensional Arrays

- Many problems require information be organized as a two-dimensional or multidimensional list
- Examples
  - Matrices
  - Graphical animation
  - Economic forecast models
  - Map representation
  - Time studies of population change
  - Microprocessor design
Multidimensional Arrays

- We can create arrays of arrays
  - Ex. an array of student grades: each array element (a student) consists of an array (of grades)
  - Use a set of square brackets for each dimension:
    ```
    int[][] grades = new int[120][6];
    ```

Multidimensional Arrays

- Not every array element must be the same length; a multidimensional array can be “ragged”
- A multidimensional array can be initialized upon creation
  - Use a list of initializers:
    ```
    int[][] foo = { {1, 2, 3} , {4, 5, 6} , {7, 8, 9} };  
    ```

Example

- Segment
  ```
  int[][] m = new int[3][];
  m[0] = new int[4];
  m[1] = new int[4];
  m[2] = new int[4];
  ```
  - Produces
Example
• Segment
  for (int r = 0; r < m.length; ++r) {
    for (int c = 0; c < m[r].length; ++c) {
      System.out.print("Enter a value: ");
      m[r][c] = stdin.nextInt();
    }
  }

Example
• Segment
  String[][] s = new String[4][];
  s[0] = new String[2];
  s[1] = new String[2];
  s[2] = new String[4];
  s[3] = new String[3];
• Produces

A Sample Array Problem
๏ Write a method that examines two integer arrays representing lottery numbers (one contains the winning numbers, and one contains a given player’s selected numbers). The method returns the number of values that appear in both arrays (i.e., the intersection of the two arrays). You may assume that the arrays are of equal length and that each array only contains one copy of a single number.
๏ Ex. \{7, 4, 9, 1, 3\} and \{4, 2, 9, 7, 3\} have four values in common (3, 4, 7, and 9)
Example: Multidimensional Arrays

- A **magic square** is an \( N \times N \) grid of integers where every row, column, and major diagonal adds up to the same value.
- To create a magic square with an odd value of \( N \), use the following algorithm:
  1. Write the value 1 in the center cell of the top row
  2. Place each subsequent value (going up to \( N^2 \)) in the cell that is one row above and one column to the left of the previous value. If you run off the edge of the grid, "wrap around" to the opposite side.
  3. If the next available cell is occupied, use the cell directly below the previous value instead, then resume as in (2).

Command-Line Arguments

- **main()** takes an array of Strings as its argument
- These Strings are provided by the operating system when a program is run
  ```
  java myProg arg1 arg2 arg3
  ```
- Command-line arguments can be used to change what a program does
  - i.e., the names of files to work with

The ArrayList Class

- ArrayList is a Java **collection** that removes some of the limitations of arrays
- An array’s size is set at the time it’s created; an ArrayList can grow to hold more items
- An ArrayList still maintains constant-time insertion and access

Creating an ArrayList

- ArrayList\(<T>\) ()
  - Creates an ArrayList of type T with space for 10 elements
    - T must be an object type!
    - Ex. ArrayList\(<\text{String}>\)()
- ArrayList\(<T>\) (int n)
  - Creates an ArrayList of type T with space for n elements
Examples

ArrayList<String> city = new ArrayList<String>();
ArrayList<String> bush = new ArrayList<String>(20);
ArrayList<Integer> number = new ArrayList<Integer>();

Adding Elements

- add (T v)
  - Adds v to the end of the ArrayList
- add (int i, T v)
  - Inserts v at position i in the ArrayList
  - Shifts elements at i+1... down one space

Insertion Examples

city.add(“Madrid”);
city.add(“Cairo”);
bush.add(“Forsythia”);
bush.add(“Azalea”);
number.add(2);
number.add(0,3); // puts 3 in position 0

Access Methods

- get (int i)
  - Returns the value at position i
  - Throws exception if i is not valid
- remove (int i)
  - Removes and returns value at position i
  - Shifts elements i+1... left to fill space
Access Examples

String value = city.get(1);
System.out.print(city.get(0));

int temp = 2 * number.get(0).intValue();
number.remove(1);

Other Useful Methods

- size()
  - Returns # of elements in the ArrayList
- clear()
  - Removes all elements from the ArrayList